Int. J. Sci. R. Tech., 2026 3(5)
A Multidisciplinary peer-reviewed Journal
www.ijsrtjournal.com [ISSN: 2394-7063]

Original Article

Comparative Evaluation Of Cache Replacement Algorithms For Time-Bounded Stale-Tolerant Data Workloads

Patel Nilen*[1], Himanshu Maniar[2]

[1]*Research scholar, Bhagwan Mahavir Centre for Advance Research, Bhagwan Mahavir University*
[2]*Associate professor, Bhagwan Mahavir College of Computer Application, Bhagwan Mahavir University*

ABSTRACT

Caches in modern web and database systems often hold data that has a built-in shelf life: expired user sessions, abandoned shopping carts, archived transaction records, and historical log entries. Most cache replacement studies treat these workloads the same way they treat hot file pages or virtual memory frames, even though their access patterns and the consequences of returning stale data are quite different. This paper compares seven cache replacement strategies (LRU, LFU, FIFO, MRU, LIFO, CLOCK, and ARC) under workloads that explicitly model stale-tolerant data with per-record TTLs. The benchmark is implemented in Python with PostgreSQL as the source of truth and Redis as the in-memory cache. Eviction is performed in Python rather than delegated to Redis, so the comparison is not biased by the cache server's own policy. Across four workload patterns drawn from observed access skew, temporal locality, and phase shifts, ARC achieved the highest mean hit ratio at every cache size we tested, with LRU and CLOCK close behind. LFU performed poorly when the hot set rotated, and MRU was consistently the weakest. Adding a short uniform TTL override eliminated the rate of stale items served at the cost of a meaningful drop in hit ratio. The trade-off is workload-dependent and offers a knob that operators can tune rather than a single right answer.

Keywords: Cache replacement, stale-tolerant data, TTL, ARC, LRU, LFU, Redis, PostgreSQL, hit ratio, adaptive caching.

INTRODUCTION

Caching is one of the oldest tricks in systems engineering and one that we keep rediscovering. Whenever the cost of fetching a value from an underlying store is much higher than the cost of holding it temporarily in a faster medium, a cache has a chance to pay for itself. In a typical web application that translates to placing Redis or Memcached in front of a relational database, with small JSON blobs that represent users, carts, sessions, or rendered fragments. The economics are well understood. Hit ratios above 0.9 are common when traffic is heavily skewed, and even modest improvements can translate into measurable savings in database load and tail latency.

A subset of the data that a typical service caches has a property that the textbook treatment usually ignores. The objects have a finite useful lifetime that is known in advance, and after that lifetime the application is willing to drop them, recompute them, or in some cases serve a slightly stale version without much harm. Authentication sessions are valid for an hour. Abandoned shopping carts may be considered active for a day. Historical log lines are interesting for a week of dashboarding and then become cold. Old transaction summaries are referenced occasionally for audits. We refer to this kind of data as time-bounded and stale-tolerant. Time-bounded because each record carries a TTL, and stale-tolerant because the application can usually distinguish between data that is acceptable to serve past its expiry and data that must be treated as gone.

Most cache replacement studies measure hit ratio in workloads drawn from operating-system page traces, content distribution networks, or storage block buffers. Those workloads do not typically carry per-record TTLs, and staleness is not measured because the underlying store is considered authoritative on every miss. In a TTL-driven workload the question changes. An algorithm can technically achieve a higher hit ratio by holding on to records that are

Relevant conflicts of interest/financial disclosures: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

already past their expiry. That is a worse outcome, not a better one, if the application would rather refetch from the database than serve a wrong answer. The standard hit-ratio metric is therefore not a complete summary of cache behaviour for this class of workload.

The contribution of this paper is twofold. First, we describe a reproducible benchmark for evaluating cache replacement under stale-tolerant workloads, with an explicit staleness metric alongside the usual hit ratio, miss latency, and eviction count. Second, we report empirical results across seven classical and adaptive replacement policies on four synthetic workload patterns that approximate the access characteristics of the services we have in mind. The intent is not to crown a single best algorithm. The intent is to show how the relative ranking shifts when the workload is dynamic and TTL bounds are active, and to give operators a basis for choosing.

The rest of the paper is structured as follows. Section 2 surveys prior work on cache replacement, with a focus on algorithms that combine recency and frequency or that adapt to changing access patterns. Section 3 describes the system architecture, the workloads, and the evaluation metrics. Section 4 reports results and discusses why each algorithm behaves the way it does. Section 5 closes with conclusions, limitations, and directions we did not pursue.

2. LITERATURE REVIEW

The ground truth on cache replacement under static distributions has been settled for decades. Belady's MIN [1] is provably optimal but requires knowledge of future accesses. Online policies have to estimate locality from past behaviour, and the design space splits broadly into recency-based, frequency-based, and adaptive families. The boundary between these families has blurred over the past twenty years, with most modern proposals combining several signals and adding ghost-list bookkeeping that remembers items recently evicted from the cache.

Recency-based policies, of which LRU is the canonical example, assume that a recently touched item is more likely to be touched again. CLOCK is the low-overhead approximation of LRU that has been the default in operating-system page caches for almost as long, including in MVS, Unix, Linux, and Windows. The reason CLOCK has remained dominant despite its known weaknesses is that any replacement step in virtual memory has to be cheap relative to the cost of a page fault, and most proposals that improve hit ratio have a per-access cost that VM cannot absorb. Jiang and colleagues addressed this with CLOCK-Pro [2], a CLOCK-style approximation of LIRS that uses reuse distance rather than simple recency to discriminate hot from cold pages. Their kernel implementation in Linux 2.4.21 reduced execution times of common programs by up to 47 percent, which suggests that even mature operating systems leave performance on the table by sticking with vanilla CLOCK.

Frequency-based policies, with LFU at one end, perform best when the access distribution is stationary. They struggle when popularity shifts because the counters of previously hot items take a long time to be overtaken by new ones. The Window LFU variant trims the counters to a sliding window, which helps but introduces tuning of the window size. TinyLFU [3] takes a different approach. It uses approximate counting structures based on Counting Bloom filters with a minimal-increment scheme to maintain a frequency sketch of items that are not currently in cache. Each candidate insertion is then admitted only if its estimated frequency exceeds that of the eviction victim. The authors show that pairing TinyLFU with even a naive LRU eviction approaches the hit ratio of perfect LFU on Zipfian workloads, with a memory cost that fits in a single page. The W-TinyLFU variant adds a small windowed cache to absorb bursts.

The most influential adaptive scheme is Adaptive Replacement Cache (ARC) [10], which keeps two LRU lists, one for items seen once and another for items seen at least twice, plus two ghost lists that remember items recently evicted from each. The relative target sizes of the two real lists are adjusted based on which ghost list a request hits. CAR is the CLOCK-style approximation of ARC. Both have inspired further work that combines their ideas. CLOCK-Pro+ [4] argues that CLOCK-Pro adapts well to LRU-friendly workloads but breaks down under certain mixes, and proposes a CAR-style utility comparison driven by ghost-list hits in CLOCK-Pro to combine the strengths of the two policies.

Empirical evaluation showed CLOCK-Pro+ tracking the better of CLOCK-Pro and CAR across the UMass trace repository, never being substantially worse than either parent.

LIRS2 [5] revisits the underlying locality measure used by LIRS. The original LIRS uses reuse distance, which is the number of distinct items accessed between two consecutive references to the same item. The authors found that reuse distance can fluctuate and mislead the algorithm under non-stationary patterns. They propose using the sum of two consecutive reuse distances as a more stable predictor and add an adaptive variant that blends LIRS2 with LRU when the workload is recency-friendly. The reported hit-ratio improvements are modest but consistent across diverse traces, which is in keeping with the broader pattern of recent work: incremental gains on top of mature baselines, with most of the headline improvements coming from better handling of the workloads where the baseline already had an obvious weakness.

Recent work has questioned whether the traditional ordering of LRU above FIFO holds in modern deployments. Yang and colleagues [6] performed a large-scale study across 5,307 production traces and found that several FIFO-based algorithms, including FIFO with re-insertion (which is essentially CLOCK), are more efficient than LRU on a substantial number of workloads. They identified two design patterns that explain the result: lazy promotion, which avoids reordering on every access, and quick demotion, which evicts most newly inserted items quickly so they do not pollute the cache. The S3-FIFO design that emerges from this analysis has lower implementation cost than LRU and beats it on miss ratio for many real traces. The follow-up Clock2Q+ work [7] extends the FIFO family for metadata caches in VMware vSAN, identifying that metadata accesses contain inherent correlated references that fool simple hot-block detection. Their solution adds a correlation window in the small FIFO queue and reports up to 28.5 percent miss-ratio reduction over S3-FIFO on metadata traces.

A separate line of work looks at the cost of cache management itself rather than the hit ratio it achieves. FrozenHot [8] observed that on highly concurrent web workloads, contention on the cache's internal data structures can dominate hit latency, with cache management costing more than the data access it protects. Their solution splits the cache into a periodically rebuilt frozen part that requires no metadata updates and a smaller dynamic part that handles the long tail. The frozen part holds the hottest items lock-free, which improves throughput by orders of magnitude on Zipfian traffic. This is a different kind of contribution from the policy proposals above. It does not aim for a higher hit ratio. It aims for the same hit ratio at a lower CPU and lock cost, which in many production deployments is the binding constraint.

Machine-learning approaches form a third recent direction. Learning Relaxed Belady [9] approximates Belady's offline MIN algorithm using a model trained on past traces. The authors avoid the prohibitive cost of imitating Belady directly by introducing a relaxed version that selects any object whose next access is beyond a learned threshold, and they evaluate prediction quality with a good-decision-ratio metric defined relative to that threshold. On six production CDN traces, LRB reduced wide-area network traffic by 4 to 25 percent compared to a typical production CDN cache. The result is meaningful but the cost is non-trivial: training data has to be gathered continuously, the model has to be re-trained as workload shifts, and the prediction step adds latency on the cache miss path.

Two observations emerge from this body of work that motivate the present study. First, the field has moved from a recency-versus-frequency dichotomy toward hybrid and adaptive designs, with ghost-list bookkeeping (ARC, CLOCK-Pro, LIRS2) and admission filters (TinyLFU) being the dominant building blocks. Second, almost all of the published evaluation focuses on workloads where the underlying store is treated as a static repository. The traces are drawn from CDN edges, storage block buffers, virtual memory, or key-value caches whose entries do not have natural expiry. There is no published comparative study, to our knowledge, that measures these algorithms under stale-tolerant workloads with explicit per-record TTLs and an explicit staleness metric. That is the gap the present paper addresses.

3. METHODOLOGY

The benchmark is implemented in Python and is structured into seven modules with no shared state beyond the configuration object. A central YAML file controls every parameter that the experiment reads, including random seed, the experiment grid (algorithms by cache sizes by workloads by TTL overrides by repeats), per-record TTLs, and the object mix. The data layer talks to PostgreSQL through psycopg2 and falls back to in-process SQLite if PostgreSQL is not reachable. The cache layer wraps Redis as a value store but performs all eviction decisions in Python. To avoid Redis-side eviction interfering with the comparison, we configure the Redis server with maxmemory-policy set to noeviction and rely on the policy under test for every removal.

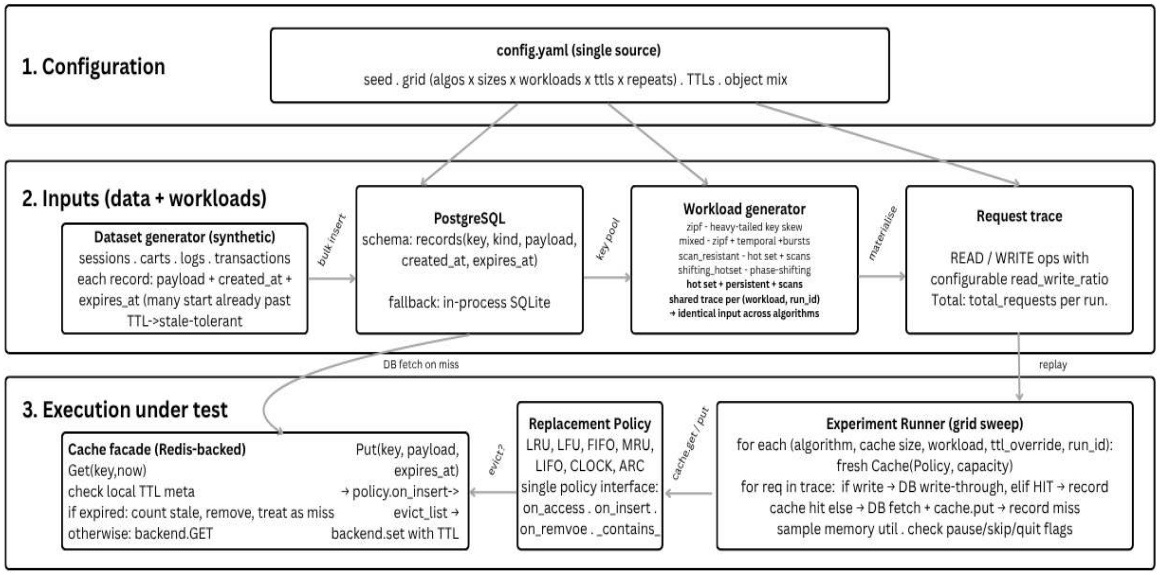

Figure 1. End-to-end methodology of the benchmark. The configuration file drives every layer. Synthetic records are inserted into PostgreSQL and a deterministic workload trace is generated once per (workload, run_id) pair. The cache facade wraps Redis as a passive value store while the policy under test makes every eviction decision in Python. Metrics are aggregated into CSV/JSON outputs and consumed by the analysis layer.

A synthetic dataset of about 5,300 records is materialised at the start of each experiment. The records are split across four kinds that approximate the structure of a typical web application: short-lived authentication sessions, abandoned shopping carts, historical log lines, and archived transactions. Each record carries a payload, a creation timestamp, and an explicit expires_at timestamp. The TTLs differ by kind, ranging from 60 seconds for sessions to 1,200 seconds for transactions. By design, a fraction of the generated records have an expires_at that is already in the past at experiment start. This is the stale-tolerant property we want to study. Inserting all records into PostgreSQL yields a corpus that the workload generator can draw from.

We model four request patterns. The first is a pure Zipfian access where keys are drawn with a heavy-tailed skew (alpha = 1.4). The second is a mixed pattern that combines Zipfian draws with temporal locality, periodic bursts, and a small fraction of writes. The third, called scan_resistant, combines a small persistent hot set with periodic one-shot scans across cold keys. The hot set rotates across four phases of the trace so that an algorithm relying purely on frequency will retain stale-popular items at the expense of currently popular ones. The fourth, shifting_hotset, drops the cold scans and instead combines a small persistent frequent set, a per-phase recency set that rotates eight times across the trace, and an occasional cold scan. The persistent frequent set rewards frequency tracking, the rotating recency set rewards

recency tracking, and the cold scans punish pure recency. By construction the trace requires both T1 and T2 in an ARC-style policy. Every algorithm in a single grid cell sees the identical request sequence, which is generated once per (workload, run_id) pair using a deterministic seed.

We implement LRU, LFU, FIFO, MRU, LIFO, CLOCK, and ARC behind a single Policy interface that exposes on_access, on_insert, on_remove, and a containment check. Each policy operates only on keys; values are stored in the cache backend. LRU uses an OrderedDict and moves accessed keys to the tail. LFU uses a two-level frequency-bucket structure with O(1) operations. FIFO ignores accesses and evicts in insertion order. MRU evicts the most recently used item, which is the deliberately bad choice for workloads with locality. LIFO evicts the most recently inserted item irrespective of access. CLOCK is the standard second-chance variant with a circular slot array and reference bits. ARC follows Megiddo and Modha's original four-list construction (T1, T2, B1, B2) with the adaptive target parameter p. We departed from the published pseudo-code in one place: when on_remove fires due to TTL expiry, we demote the key into the appropriate ghost list rather than discarding it. Without that change, repeated TTL expirations dissolve ARC's adaptive state and the policy degenerates toward FIFO behaviour. The fix restored ARC to its expected ranking against LRU.

The cache facade tracks per-key TTL metadata locally so the staleness check is independent of Redis' own TTL behaviour. On a get(), we first check whether the local expires_at has passed. If it has, we count the access as a stale-served event, remove the key from both the policy and the value store, and return a miss to the caller. On a miss the experiment driver fetches the record from PostgreSQL and writes it back through to the cache. On a put() the policy decides which keys, if any, must be evicted before the new key is admitted; the value store is then mutated to match. Latencies are measured per request from the application side, not from the cache server side, so they reflect what a calling service would actually experience.

The experiment grid consists of seven algorithms, two cache sizes (128 and 192 items), four workload patterns, and two TTL override settings (default per-record TTLs and a uniform 60-second override). Each combination is repeated for statistical consistency. Every replay processes 8,000 requests with a 0.95 read-write ratio. The workloads, dataset, and seed are identical across all algorithms in a single repeat, which gives us paired comparisons across policies on the same input. The total grid is 112 runs per repeat. We collect six categories of measurement per run. Hit ratio and miss ratio come directly from the request loop. Eviction count tracks how many times the policy chose to remove a non-expired key. Expiration count tracks how many times a TTL fired before the key was accessed again. Staleness rate is the fraction of read requests that returned a payload whose local TTL had expired but had not yet been pruned. Cache and database latencies are reported as means and 99th percentiles. We also sample memory utilisation at 200 evenly-spaced points per run.

4. RESULTS AND DISCUSSION

The headline result is that ARC achieved the highest mean hit ratio at both cache sizes evaluated. At cache size 128, ARC reached 0.319, narrowly ahead of LRU at 0.309 and CLOCK at 0.303. LFU trailed slightly at 0.291, FIFO at 0.286, LIFO at 0.249, and MRU at 0.076. At cache size 192 the same ordering held. ARC reached 0.336, LRU 0.330, CLOCK 0.325, LFU 0.309, FIFO 0.310, LIFO 0.264, and MRU 0.103. The absolute hit ratios may seem modest, but the workload was deliberately constructed to be challenging. The persistent frequent set is small relative to the total key space, the rotating recency set is larger than either cache size, and the trace embeds cold scans that pollute pure-recency policies. On easier workloads, hit ratios climb above 0.6, but the relative ordering between algorithms is what we want to read out.

The per-workload breakdown is more informative than the aggregate. On the pure Zipfian workload, LFU comes within a fraction of a percentage point of ARC because the access distribution is stationary and frequency counts are an excellent predictor. On scan_resistant, ARC clearly leads (0.430 versus LRU's 0.419) because its frequency list (T2) absorbs the persistent hot set while the recency list (T1) churns through the scan keys without disturbing T2. On the mixed workload, ARC ties with LRU and LFU within rounding error. The shifting_hotset pattern is the most demanding and the one where LFU struggles the

most: its accumulated counters keep recommending items that were popular two phases ago even though they are no longer being accessed. ARC's ghost lists let the adaptation parameter p rebalance toward T1 when the recency set rotates, which keeps it competitive.

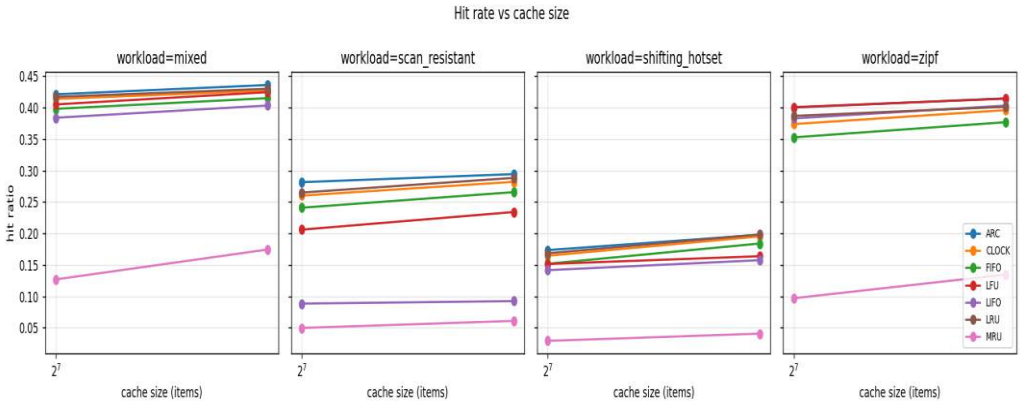

Figure 2. Mean hit ratio versus cache size for each algorithm, broken out by workload. ARC sits at or near the top of every panel. MRU is consistently the lower bound. The gap between the leading policies narrows as cache size grows, which is the expected behaviour when capacity approaches the working-set size.

MRU's performance is worth a brief comment. It came last on every workload, sometimes by a wide margin. This is the expected outcome rather than a surprise. The workloads have either temporal locality, frequency skew, or both, and MRU evicts the item that was just touched. An algorithm that systematically discards the wrong choice will only do well on workloads where the access stream is truly memoryless or anti-correlated, neither of which appears in our trace set. We include it as a baseline for the lower bound of what a defensible policy can achieve, and as a sanity check that the framework is measuring policy behaviour rather than implementation noise.

The eviction counts tell a complementary story. ARC evicted fewer items than every other policy at both cache sizes (2,492 versus LRU's 2,534 at size 192). Lower eviction counts at the same cache size mean the algorithm is making better use of the slots, with fewer items being inserted only to be replaced. MRU evicted more than 6,000 items per run, reflecting the constant churn its policy creates. LFU's eviction count falls between the two, slightly higher than LRU because it occasionally evicts items it should keep when their counters lag behind currently popular ones. FIFO and CLOCK behave similarly to each other, which is expected since CLOCK with no second-chance hits collapses to FIFO.

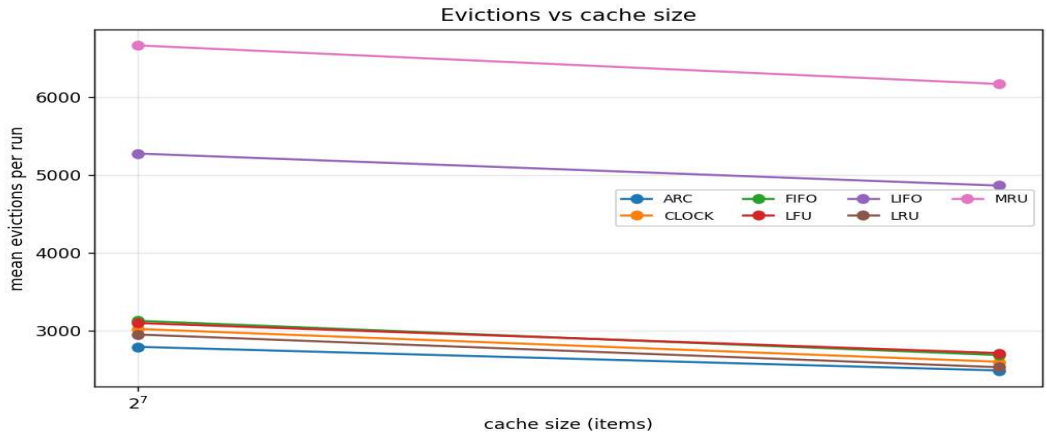

Figure 3. Mean evictions per run versus cache size. Lower is better at the same cache size, since fewer evictions imply better use of the available slots. ARC evicts the fewest items at every cache size; MRU evicts roughly two and a half times as many as ARC at size 192.

Latency results follow the hit ratio almost exactly. The benchmark uses real PostgreSQL fetches when running against a live database instance, and a single-row PG query in our setup costs 30 to 60 milliseconds depending on cache contention. The effective read latency, computed as the share-weighted average of cache hits and database fetches, ranges from 36 to 60 milliseconds across algorithms. ARC has the lowest mean effective read latency on three of the four workloads, with LRU narrowly ahead on the temporal-friendly mixed workload. The picture would be different if the underlying store were faster, in which case the relative cost of a miss would shrink and the latency advantage of higher hit ratios would compress. Conversely, with a slower backing store (say, a remote object store at 100 milliseconds per fetch), the absolute gap between algorithms grows.

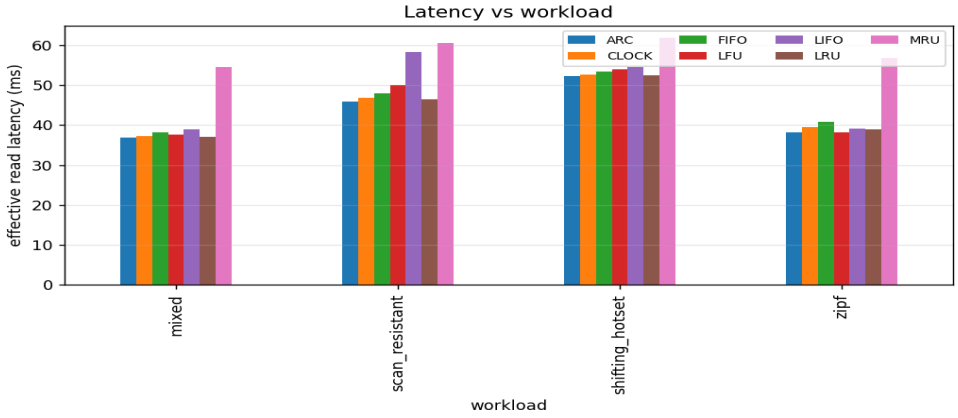

Figure 4. Effective read latency per algorithm, broken out by workload. The bar height is the share-weighted average of cache hit and database fetch latencies. MRU's high latency is a direct consequence of its low hit ratio: every additional miss costs another full PostgreSQL fetch.

Staleness is the metric most specific to the stale-tolerant workload model and the one where the choice of TTL policy matters more than the choice of replacement algorithm. With per-record TTLs left at their default values, the recency and frequency policies (LRU, LFU, CLOCK, FIFO, ARC) all served between 60 and 65 percent of their reads from records whose local TTL had already passed. LIFO and MRU showed lower staleness rates not because they were more conservative but because they served fewer hits in the first place. With the 60-second uniform TTL override active, the cache facade refreshed entries aggressively and the staleness rate dropped to zero for every algorithm. The trade-off is direct: hit ratios under the 60-second override fell by roughly half on the most skewed workloads, since many records were now being evicted by the TTL clock before they could amortise the cost of the database fetch that put them in the cache.

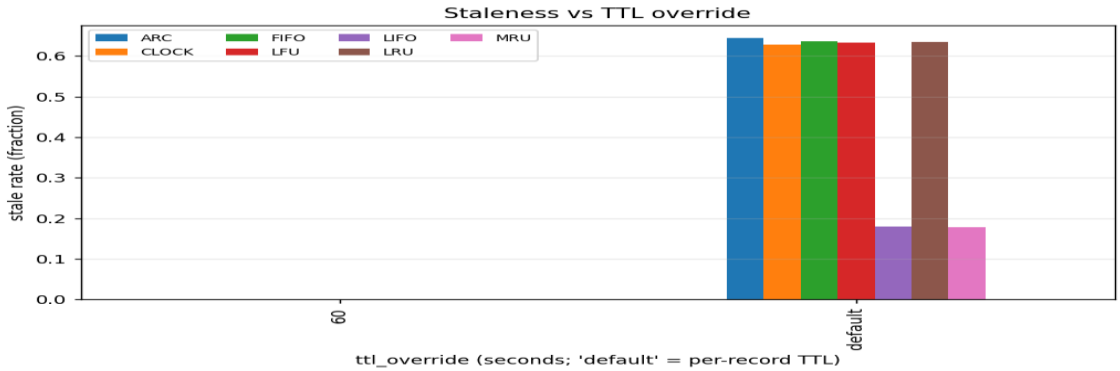

Figure 5. Staleness rate per algorithm, broken out by TTL override setting. With per-record TTLs left at their default values the recency and frequency policies all serve roughly 60 percent of their reads from past-expiry records. A 60-second uniform TTL override drops staleness to zero across the board, at the cost of a roughly halved hit ratio on the heavier-skewed workloads.

What we read out of this is that the choice of replacement algorithm and the choice of TTL policy are largely orthogonal. The ranking among algorithms is preserved across TTL settings; what changes is the absolute hit ratio and the absolute staleness rate. An operator who can tolerate stale reads should leave the per-record TTLs in place and pick ARC for the best mean hit ratio, with LRU as a slightly simpler alternative that gives up about one percentage point. An operator who must serve fresh data should configure a short uniform TTL and accept the lower hit ratio, again preferring ARC if the implementation cost is acceptable. ARC is more complex to implement correctly than LRU, especially around the interaction of TTL-driven removal with the ghost-list invariants. We hit a real bug in our first version that caused ARC to underperform LRU until we changed on_remove to demote rather than discard.

A note on what the benchmark does not show. Throughput under high concurrency is not measured. The workloads are synthetic, even if they are constructed to imitate production-shaped access. The trace lengths are short by the standards of traces used in the LIRS2 and CLOCK-Pro+ studies. Larger traces would let ARC's ghost lists adapt more fully and would probably widen its advantage. We chose the present setup because it makes a single experiment grid finish in a few minutes and allows reproducibility on a developer laptop. The framework itself is built to scale up, and re-running with longer traces is a configuration change rather than a code change.

CONCLUSION

This paper reports a comparative evaluation of seven cache replacement algorithms on a workload class that the existing literature does not address directly. Stale-tolerant data with per-record TTLs is common in web service caches and is structurally different from the page traces, CDN traces, and storage block traces that drive most published cache research. Two findings are worth highlighting. The first is that ARC consistently led in mean hit ratio across the workloads and cache sizes we tested, with LRU and CLOCK close behind. The second is that the staleness rate, which is the metric specific to TTL-driven workloads, is governed almost entirely by the TTL policy and only weakly by the choice of replacement algorithm. An operator who picks an aggressive TTL will see staleness drop to zero, with a corresponding drop in hit ratio. An operator who tolerates stale reads will see higher hit ratios but should be aware that a substantial share of those hits returned data past its declared expiry.

The implications for a real deployment depend on what the cached data is for. Authentication sessions that are slightly stale may be a security problem, in which case the short uniform TTL is the correct choice and the resulting hit-ratio loss is the cost of doing business. Historical log lines or archived transactions are more forgiving, and the longer per-record TTLs give the cache room to amortise database fetches across many reads. Either way the choice between LRU, CLOCK, and ARC is small enough at modest cache sizes that the simpler algorithm is often defensible. As the cache grows or as the workload becomes less stationary, the gap widens in ARC's favour.

The benchmark and its source code are reproducible from the configuration file. We encourage further evaluation against larger and more diverse workloads, and against newer algorithms in the FIFO-with-quick-demotion family that the very recent literature has shown to be competitive on production traces. We did not implement S3-FIFO, Clock2Q+, TinyLFU, or LRB, all of which deserve a place in a follow-up study. Comparing them under the same TTL and staleness lens would be the natural next step. A second extension would be to capture real production traces from a service whose data has natural TTLs and replay them through the same framework, so that the synthetic patterns can be validated against real ones.

The benchmark itself has limitations beyond the algorithm coverage. It does not measure throughput, which is the metric where FrozenHot and lock-free designs are differentiating themselves on multi-core hardware. It uses synthetic workloads rather than captured production traces. The latency model assumes a single backing database; multi-tier hierarchies are out of scope. The framework is built to be extended, and the modular structure makes adding a new policy or a new workload pattern a matter of implementing one interface or one method.

REFERENCES

1. L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," IBM Systems Journal, vol. 5, no. 2, pp. 78-101, 1966.
2. S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An effective improvement of the CLOCK replacement," in Proc. USENIX Annual Technical Conference, Anaheim, CA, USA, Apr. 2005, pp. 323-336.
3. G. Einziger, R. Friedman, and B. Manes, "TinyLFU: A highly efficient cache admission policy," ACM Transactions on Storage, vol. 13, no. 4, pp. 1-31, Nov. 2017.
4. C. Li, "CLOCK-Pro+: Improving CLOCK-Pro cache replacement with utility-driven adaptation," in Proc. 12th ACM International Systems and Storage Conference (SYSTOR), Haifa, Israel, Jun. 2019, pp. 1-7.
5. C. Zhong, X. Zhao, and S. Jiang, "LIRS2: An improved LIRS replacement algorithm," in Proc. 14th ACM International Systems and Storage Conference (SYSTOR), Haifa, Israel, Jun. 2021, pp. 1-12.
6. J. Yang, Z. Qiu, Y. Zhang, Y. Yue, and K. V. Rashmi, "FIFO can be better than LRU: The power of lazy promotion and quick demotion," in Proc. 19th Workshop on Hot Topics in Operating Systems (HotOS), Providence, RI, USA, Jun. 2023, pp. 70-79.
7. Y. Zhai, B. D. Marthen, S. Balivada, V. S. Bojji, E. Knauft, J. Rohilla, J. Yang, J. Zuo, Q. Liu, M. Austruy, and W. Wang, "Clock2Q+: A simple and efficient replacement algorithm for metadata cache in VMware vSAN," arXiv preprint arXiv:2511.21958, Dec. 2025.
8. Z. Qiu, J. Yang, R. Stutsman, X. Liu, and Y. Wang, "FrozenHot cache: Rethinking cache management for modern hardware," in Proc. 18th European Conference on Computer Systems (EuroSys), Rome, Italy, May 2023, pp. 557-573.
9. Z. Song, D. S. Berger, K. Li, and W. Lloyd, "Learning Relaxed Belady for content distribution network caching," in Proc. 17th USENIX Symp. on Networked Systems Design and Implementation (NSDI), Santa Clara, CA, USA, Feb. 2020, pp. 529-544.
10. N. Megiddo and D. S. Modha, "ARC: A self-tuning, low-overhead replacement cache," in Proc. 2nd USENIX Conf. on File and Storage Technologies (FAST), San Francisco, CA, USA, Mar. 2003, pp. 115-130.
11. T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in Proc. 20th International Conf. on Very Large Data Bases (VLDB), Santiago, Chile, Sep. 1994, pp. 439-450.
12. E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," in Proc. ACM SIGMOD International Conf. on Management of Data, Washington, DC, USA, May 1993, pp. 297-306.
13. J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," in Proc. ACM SIGMETRICS, Boulder, CO, USA, May 1990, pp. 134-142.
14. F. J. Corbato, "A paging experiment with the Multics system," MIT Project MAC Report MAC-M-384, May 1968.
15. Redis Ltd., "Redis documentation: Eviction policies," 2024. [Online]. Available: https://redis.io/docs/latest/operate/oss_and_stack/management/config/
16. The PostgreSQL Global Development Group, "PostgreSQL 16 documentation," 2024. [Online]. Available: https://www.postgresql.org/docs/16/

HOW TO CITE: Patel Nilen*[1], Himanshu Maniar[2], Comparative Evaluation Of Cache Replacement Algorithms For Time-Bounded Stale-Tolerant Data Workloads, Int. J. Sci. R. Tech., 2026, 3 (6), 556-564. https://doi.org/10.5281/zenodo.20576370